# Класи- основа об’єктно-орієнтованого програмування

# ЗМІСТ

1. Базові поняття класу
2. Доступ до членів класу
3. Конструктори й деструктори
4. Класи й структури – споріднені типи
5. Об'єднання й класи - споріднені типи
6. Поняття про вбудовані функції
7. Особливості організації масивів об'єктів

# ЛІТЕРАТУРА:


1. Бублик В.В. Об’єктно-орієнтоване програмування: [Підручник] / В.В. Бублик. – К.: ІТ-книга, 2015. – 624 с.
2. Вступ до програмування мовою С++. Організація обчислень : навч. посіб. / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с.
3. Зубенко В.В., Омельчук Л.Л. Програмування. Поглиблений курс. – К.:Видавничо-поліграфічний центр “Київський університет”, 2011. - 623 с.
4. Страуструп Бьярне. Программирование: принципы и практика с использованием С++, 2-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильяме", 2016. - 1328 с.
5. Прата С. Язык программирования С++. Лекции и упражнения. Учебник. -СПб. ООО «ДиаСофтЮП», 2003. 1104 с.
6. Шилдт Г. С++: базовый курс, 3-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2010. – 624 с.
7. Stroustrup, Bjarne. The C++ programming language. — Fourth edition. – Addison-Wesley, 2013. – 1361 pp.

# Основи поняття класу

**Клас визначає новий тип даних**, що задає формат об'єкта.

**Клас включає (інкапсулює) як дані, так і код, призначений для опрацювання цих даних.**

**Клас** зв'язує дані з кодом.

Специфікація класу використовується для побудови об'єктів.

**Об'єкти - це екземпляри класу.**

Клас - це логічна абстракція (проект об’єктів), що реально не існує доти, поки не буде створений об'єкт цього класу.

**Функції й змінні, що складають клас, називаються його членами.**

Змінна, оголошена в класі, називається **членом даних (змінною екземпляра або змінною реалізації),** а функція, оголошена в класі, називається **функцією-членом** (методом).

# Основи поняття класу

Оголошення класу починається словом class.

```
class queue {
   int q[100];        // масив під елементи черги
   int sloc, rloc;    // індекси кінця і початку черги
  public:
   void init();  // ініціалізатор черги
   void qput(int i);     // занесення елемента в чергу
   int qget();   // добування елемента з черги
};
```

Клас може містити як закриті, так і відкриті члени.

**За замовчуванням всі елементи, визначені в класі, є закритими.**

Всі змінні або функції, визначені після специфікатора *public*, доступні для всіх інших функцій програми. У програмі **доступ до закритих членів класу організується через його відкриті функції.** Кожна функція-член має доступ до закритих елементів класу.

# Екземпляр класу (змінна класу, об’єкт)

Ім'я класу є специфікатором нового (обєктного) типу.
Інструкція оголошення:

```
queue Q1, Q2;
```

створює два об'єкти *Q1* і *Q2* типу queue, кожному з яких виділяється окрема ділянка пам’яті з власними копіями членів даних*: q, sloc, rloc*; та точками входу у функції-члени: *init(), qput() і qget*().

Реалізації функцій-членів програмуються окремо, наприклад:

```
void queue::qput(int i) {
  if(sloc==100) {
    cout << "Черга заповнена.\n";
    return;
  }
  sloc++;
  q[sloc] = i;
}
```

# Екземпляр класу (змінна класу, об’єкт)

Оператор дозволу області видимості “::” кваліфікує ім'я члена разом з ім'ям його класу , тобто оголошує компілятору , що дана версія функції qput() належить класу queue.

**Різні класи можуть використовувати однакові імена функцій.**

Скомпільований код функціїї-члена певного класу знаходиться серед кодів програми і використовується сумісно всіма об’єктами цього класу через свої точки входу.

**Щоб викликати відкритий член даних або відкриту функцію-член із частини програми, що перебуває поза класом, необхідно використовувати ім'я об'єкта й оператор "крапка“**:

```
Q1.init();
```

Функція-член може звертатись до будь-якого члена даних

```
#include <iostream>
using namespace std;
class queue { // Створення класу
    int q[100];
    int sloc, rloc;
  public:
    void init();
    void qput(int i);
    int qget();
};
// Ініціалізація класу queue
void queue::init() {
  rloc = sloc = 0;
}
// Занесення в чергу значення
void queue::qput(int i) {
  if(sloc==100) {
    cout << "Черга заповнена.\n";
    return;
  }
  sloc++;
  q[sloc] = i;
}
```

```
// Добування із черги значення
int queue::qget() {
  if(rloc == sloc) {
    cout << "Черга порожня.\n";
    return 0;
  }
  rloc++;  return q[rloc];
}
int main() {
  queue a, b;
  a.init();    b.init();
  a.qput(10);  b.qput(19);
  a.qput(20);  b.qput(1);
  cout << "Вміст черги a: ";
  cout << a.qget() << " ";
  cout << a.qget() << "\n";
  cout << "Вміст черги b: ";
  cout << b.qget() << " ";
  cout << b.qget() << "\n";
  return 0;
}
```

Програма генерує такі результати:

```
Вміст черги a: 10 20
Вміст черги b: 19 1
```

# Загальний формат оголошення класу

```
class ім'я_класу {
    закриті дані й функції
  public:
    відкриті дані й функції
} список_об'єктів;
```

Або

```
class ім'я_класу {
  private:
     закриті дані та функції класу
  public:
                 відкриті дані та функції класу
};
ім'я_класу перелік_об'єктів_класу;
```

# Загальний формат оголошення класу

**Варто знати ! Об'єкт утворює зв'язки між різними частинами коду програми і її даними.**

Будь-яка функція-член класу має доступ до закритих елементів класу.

Це означає, що функції init(), qget() і qput() мають доступ до змінних rloc та sloc.

Щоб **додати будь-яку функцію в клас**, необхідно оголосити її прототип у визначенні цього класу, після чого вона стає функцією-членом класу.

# ОЗНАЧЕННЯ:

- **Клас** в **C++** - це визначений користувачем тип або структура даних, оголошена ключовим словом *class*, яка містить дані (поля) і функції (методи) як свої члени, доступ до якої регулюється трьома специфікаторами доступу: *private, public, protected*.

# Доступ до членів класу

**Правило:** **коли доступ до деякого члена класу відбувається ззовні цього класу, його необхідно кваліфікувати (уточнити) за допомогою імені конкретного об'єкта.**

**Код самої функції-члена може звертатися до інших членів того ж класу прямо.**

```
#include <iostream>
using namespace std;

class myclass {
    int a; // закриті члени
  public:  // відкриті члени
    int b;
    void setab(int i);
    int geta();
    void reset();
};

void myclass::setab(int i) {
  a = i; // пряме звертання
  b = i*i; // пряме звертання
}

int myclass::geta() {
  return a; //пряме звертання
}

void myclass::reset() {
  setab(0); //Прямий виклик
}
```

```
int main()
{
  myclass ob;
  ob.setab(5); // Установл. ob.a і ob.b
  cout << “ob після виклику setab(5): ";
  cout << ob.geta() << ' ';
 cout << ob.b; // b є public-членом
  cout << '\n';
 ob.b = 20; // b встановлюємо прямо
  cout << “ob після ob.b=20: “;
  cout << ob.geta() << ' ';
  cout << ob.b;
  cout << '\n';
  ob.reset();
  cout<<“ob після виклику ob.reset(): “;
  cout << ob.geta() << ' ';
  cout << ob.b;
  cout << '\n';
  return 0;
}
```

Результат роботи програми:

```
ob після виклику setab(5): 5 25
ob після ob.b=20: 5 20
ob після виклику ob.reset(): 0 0
```

# *Необхідно пам’ятати!*

- Не варто хвилюватися з приводу того, що Ви ще не відчули упевненості щодо механізму доступу до членів класу.
- Невелике Ваше занепокоєння при освоєнні цього питання – звичайне явище для програмістів-початківців.
- Сміливо продовжуйте вивчати C++, розглядаючи якомога більше прикладів, і питання доступу до членів класу незабаром стане так само простим, як таблиця множення для першокласника!

# Конструктори й деструктори

***Конструктор*** – це спеціальна функція-член класу, яка викликається при створенні об'єкта, а її ім'я обов'язково збігається з іменем класу.

```
class queue // Визначення класу queue.
{
    int q[100];
    int sloc, rloc;
  public:
    queue(); // конструктор
    void qput(int i);
    int qget();
};
```

# *Конструктори й деструктори*

В оголошенні конструктора відсутній тип значення, що повертається, оскільки конструктори не повертають значень. (Можна вказувати навіть тип void.)

```
// Визначення конструктора.
queue::queue()
{
  sloc = rloc = 0;
  cout << "Черга ініціалізована\n";
}
```

**Конструктор викликається при створенні об'єкта.**

Це означає, що він викликається при виконанні інструкції оголошення об'єкта.

Конструктори глобальних об'єктів викликаються на самому початку виконання програми, ще до звертання до функції main(). Конструктори локальних об'єктів викликаються щораз, коли зустрічається оголошення такого об'єкта.

**Деструктор - це функція-член, що викликається при руйнуванні об'єкта.**

При руйнуванні об'єкту необхідно виконати деяку дію або навіть деяку послідовність дій.
Існує багато факторів, що спричиняють необхідність деструктора. Наприклад, об'єкт повинен звільнити раніше виділену для нього пам'ять.
Ім'я деструктора збігається з ім'ям конструктора, але випереджається символом "~".
Подібно конструкторам деструктори не повертають значень, а отже, у їхніх оголошеннях відсутній тип значення, що повертається.

```
// Визначення класу queue.
class queue {
    int q[100];
    int sloc, rloc;
  public:
    queue(); // конструктор
    ~queue(); // деструктор
    void qput(int i);
    int qget();
};

queue::queue() // конструктор
{
  sloc = rloc = 0;
  cout<<"Черга ініціалізована.\n";
}
queue::~queue() // деструктор
{
  cout << "Черга зруйнована.\n";
}
```

```
#include <iostream>
using namespace std;
class queue {
    int q[100];
    int sloc, rloc;
  public:
    queue(); // конструктор
    ~queue(); // деструктор
    void qput(int i);
    int qget();
};
queue::queue() { // конструктор
  sloc = rloc = 0;
  cout<<"Черга ініціалізована.\n";
}
queue::~queue() { // деструктор
  cout << "Черга зруйнована.\n";
}
void queue::qput(int i) {
  if(sloc==100) {
    cout << "Черга заповнена.\n";
    return;
  }
  sloc++;  q[sloc] = i;
}
```

```
int queue::qget() {
  if(rloc == sloc) {
    cout << "Черга порожня.\n";
    return 0;
  }
  rloc++;
  return q[rloc];
}
int main() {
  queue a, b;
  a.qput(10);  b.qput(19);
  a.qput(20);  b.qput(1);
  cout << a.qget() << " ";
  cout << a.qget() << "\n";
  cout << b.qget() << " ";
  cout << b.qget() << "\n";
  return 0;
}
```

Програма виводить такі результати:

```
Черга ініціалізована.
Черга ініціалізована.
10 20
19 1
Черга зруйнована.
Черга зруйнована.
```

## Параметризовані конструктори

Конструктор може мати параметри. З їхньою допомогою при створенні об'єкта членам даних можна присвоїти початкові значення.

Вдосконалений клас queue приймає аргументи, які будуть служити ідентифікаційними номерами (ID) черги.

```
class queue {
    int q[100];
    int sloc, rloc;
    int who; // містить ідентифікаційний номер черги
  public:
    queue(int id); // параметризований конструктор
    ~queue(); // деструктор
    void qput(int i);
    int qget();
};

// Визначення конструктора.
queue::queue(int id)
{
  sloc = rloc = 0;
  who = id;
  cout << "Черга " << who << " ініціалізоване.\n";
}
```

# Параметризовані конструктори

***Щоб передати аргумент конструктору, необхідно зв'язати цей аргумент із об'єктом при оголошенні об'єкта***.

C++ підтримує два способи реалізації такого зв'язування.

**Перший спосіб** використовує явний виклик конструктора:

```
queue a = queue(101);
```

У цьому оголошенні створюється черга з ім'ям a, якій передається значення (ідентифікаційний номер) 101.

# Параметризовані конструктори

**Другий спосіб** використовує неявний виклик конструктора, має більш короткий запис і зручніший для використання:

```
queue a(101);
```

**Це найпоширеніший спосіб оголошення параметризованих об'єктів.**

Загальний формат передачі аргументів конструкторам.

**тип_класу ім'я_змінної(список_аргументів);**

Елемент список_аргументів являє собою список розділених комами аргументів, що передаються конструктору.

Формально між двома наведеними вище формами ініціалізації існує невелика різниця.

```
#include <iostream>
using namespace std;
class queue {
    int q[100];
    int sloc, rloc;
    int who; // ід. номер черги
  public:
    queue(int id); // конструктор
    ~queue(); // деструктор
    void qput(int i);
    int qget();
};
queue::queue(int id) {
  sloc = rloc = 0;
  who = id;
  cout << "Черга " << who
       << " ініціалізована.\n";
}
queue::~queue() {
  cout << "Черга " << who
       << " зруйнована.\n";
}
```

```
void queue::qput(int i) {
  if(sloc==100) {
    cout << "Черга заповнена.\n";
    return; }
  sloc++;  q[sloc] = i;
}
int queue::qget() {
  if(rloc == sloc) {
    cout << "Черга порожня.\n";
    return 0; }
  rloc++;  return q[rloc];
}
int main() {
  queue a(1), b(2);
  a.qput(10);  b.qput(19);
  cout << a.qget() << " ";
  cout << b.qget() << "\n";
  return 0;
}
```

Програма генерує такі результати:

```
Черга 1 ініціалізована.
Черга 2 ініціалізована.
10 20 19 1
Черга 2 зруйнована.
Черга 1 зруйнована.
```

У загальному випадку конструктору можна передавати два і більше аргументів :

```
#include <iostream>
using namespace std;
class widget {
    int i;    int j;
  public:
    widget(int a, int b);
    void put_widget();
};
widget::widget(int a, int b) {
  i = a; j = b;
}
void widget::put_widget() {
  cout << i << " " << j << "\n";
}
int main() {
  widget x(10, 20), y(0, 0);
  x.put_widget();   y.put_widget();
  return 0;
}
```

При виконанні ця програма відображає наступні результати:

```
10 20
0 0
```

***На відміну від конструкторів, деструктори не можуть мати параметрів.***

## Альтернативний варіант ініціалізації об'єкта

*Якщо конструктор приймає тільки один параметр, можна використовувати альтернативний спосіб ініціалізації членів об'єкта.*

```
#include <iostream>
using namespace std;
class myclass {
    int a;
  public:
    myclass(int x);
    int get_a();
};
```

```
myclass::myclass(int x) {
  a = x;
}

int myclass::get_a() {
  return a;
}

int main() {
  myclass ob = 4; // виклик myclass(4)
  cout << ob.get_a();
  return 0;
}
```

Інструкція оголошення

```
myclass ob = 4;
```

обробляється компілятором так, ніби вона була записана в такий спосіб:

```
myclass ob = myclass(4);
```

У загальному випадку, якщо є конструктор, що приймає тільки один аргумент, для ініціалізації об'єкта можна використовувати або варіант `ob(x)`, або варіант `ob=x`. Справа в тому, **що при створенні об’єкта конструктором з одним аргументом відбувається неявне перетворення з типу цього аргументу в тип цього класу**.

*Альтернативний спосіб ініціалізації об'єктів застосовується тільки до об’єктів, конструктори яких мають тільки один параметр.*

# Класи й структури – споріднені типи

***В C++ структура також має об'єктно-орієнтовані можливості.*** Тобто, структура також може включати дані й код, що маніпулює цими даними точно так само, як це може робити клас. В*ідповідно до формального синтаксису C++* ***опис структури створює об'єктний тип, тобто як і опис класу може мати закриті члени і функції-члени***. Єдина відмінність між C++-структурою й C++-класом полягає в тому, що ***за замовчуванням члени класу є закритими, а члени структури - відкритими***.

```
// Структура як клас
#include <iostream>
using namespace std;
struct cl {
    int get_i();
    void put_i(int j);
  private:
    int i;
};
int cl::get_i() {return i;}
void cl::put_i(int j) {i = j;}
int main() {
  cl s;
  s.put_i (10);
  cout << s.get_i();
  return 0;
}
```

```
// Тип class замість типу struct
#include <iostream>
using namespace std;
class cl {
    int i;
  public:
    int get_i();
    void put_i(int j);
};
int cl::get_i() {return i;}
void cl::put_i(int j) {i = j;}
int main() {
  cl s;
  s.put_i(10);
  cout << s.get_i();
  return 0;
}
```

# *Об'єднання й класи - споріднені типи*

C++ *об'єднання - це, по суті, той же клас, у якому всі члени даних зберігаються в одній і тій же області*. (Таким чином, об'єднання також визначає тип класу.) *Об'єднання може містити конструктор і деструктор, а також функції-члени. Звичайно ж, члени об'єднання за замовчуванням відкриті (public), а не закриті (private).*

```
#include <iostream>
using namespace std;
union u_type {
  u_type(short int a); // конструктор
  void showchars();
  short int i;
  char ch[2];
};

u_type::u_type(short int a) { i = a; }

void u_type::showchars() { cout << ch[0] << " “ << ch[1] << "\n“; }

int main() {
  u_type u (16706); // 16706 = 0x4142 (01000001 01000010) -> "A" "B“
  u.showchars();
  return 0;
}
```

# Поняття про вбудовані функції

**Функція, що вбудовується (inline function), - це невелика (за об'ємом коду) функція, код якої підставляється в те місце програми, з якого вона викликається, тобто виклик такої функції заміняється її кодом.**

Метою використання функцій, що вбудовуються, є ефективність, оскільки заміна виклику функції її кодом ліквідує системні витрати на організацію самого виклику (збереження і відновлення контексту, розміщення локальних даних).

Існує два способи створення функції, що вбудовується.

## Використання модифікатора inline

Модифікатор **inline** перед оголошенням функції приписує компілятору її вбудовувати.

```
#include <iostream>
using namespace std;
class cl {
    int i; // закритий член
  public:
    int get_i();
    void put_i(int j);
};
```

```
inline int cl::get_i() {return i;}
inline void cl::put_i(int j) {i = j;}
int main() {
  cl s;
  s.put_i(10);        // s.i = 10;
  cout << s.get_i(); // cout << s.i;
  return 0;
}
```

Тут замість виклику функцій `get_i()` і `put_i()` підставляється їхній код. Так, у функції `main()` рядок `s.put_i(10);` функціонально еквівалентний інструкції присвоювання `s.i=10;` Оскільки змінна `i` за замовчуванням закрита в рамках класу `cl`, цей рядок не може реально існувати в коді функції `main()`, але за рахунок вбудовування функції `put_i()` досягається той же результат без витрат системних ресурсів, пов'язаних з викликом функції.

## Використання функцій, що вбудовуються, у визначенні класу

Функція, що визначається в оголошенні класу, автоматично стає вбудованою. У цьому випадку необов'язково випереджати її оголошення ключовим словом `inline`. Наприклад, попередню програму можна переписати так:

```
#include <iostream>
using namespace std;
class cl {
    int i; // закритий член за замовчуванням
  public:
    // функції, що вбудовуються автоматично
    int get_i() { return i; }
    void put_i(int j) { i = j; }
} s ;

int main()
{
  s.put_i(10); // генерується код з інструкції s.i = 10;
  cout << s.get_i(); // генерується код з інструкції cout << s.i;
  return 0;
}
```

***Визначення невеликих функцій-членів в оголошенні класу - звичайна практика в С++-програмуванні.***

# *Масиви об'єктів*

```
#include <iostream>
#define N 3
using namespace std;
enum resolution{low,medium,high}
class display {
  int width, height;
  resolution res;
 public:
  void set_dim(int w, int h){
    width=w; height=h;
  }
  void get_dim(int &w, int &h){
    w=width; h=height;
  }
  void set_res(resolution r){
    res = r;
  }
  resolution get_res(){
    return res;
  }
};
char names[N][8] = {"низька",
  "середня“, "висока”};
```

```
int main() {
 display display_mode[N];
 int i, w, h;
 display_mode[0].set_res(low);
 display_mode[0].set_dim(640, 480);
 display_mode[1].set_res(medium);
 display_mode[1].set_dim(800, 600);
 display_mode[2].set_res(high);
 display_mode[2].set_dim(1600, 1200);
 cout << “Режими відображення:\n";
   for(i=0; i<N; i++) { cout <<
     names[display_mode[i].get_res()];
    cout << ":";
    display_mode[i].get_dim(w, h);
    cout << w << " x " << h << "\n";
   }
 return 0;
}
```

Програма генерує такі результати:

```
Режими відображення:
низька: 640 x 480
середня: 800 x 600
висока: 1600 x 1200
```

# Ініціалізація масивів об'єктів

Якщо клас включає параметризований конструктор, що приймає один параметр, то масив об'єктів такого класу можна ініціалізувати так:

```
#include <iostream>
using namespace std;
class samp {
    int a;
  public:
    samp(int n) { a = n; }
    int get_a() { return a; }
};
int main()
{
  samp sampArray[4] = { -1, -2, -3, -4 };
  int i;
  for(i=0; i<4; i++) cout << sampArray[i].get_a() << ' ';
  cout << "\n";
  return 0;
}
```

Результати виконання цієї програми: `-1 -2 -3 -4`

Синтаксис ініціалізації масиву, поданий рядком

```
samp sampArray[4] = { -1, -2, -3, -4 };
```

реально являє собою скорочений варіант довгого формату:

```
samp sampArray[4] = { samp(-1), samp(-2), samp(-3), samp(-4) };
```

При ініціалізації масиву об'єктів, конструктори яких приймають кілька аргументів, необхідно використовувати довгий формат ініціалізації:

```
#include <iostream>
using namespace std;
class samp {
    int a, b;
  public:
    samp(int n, int m) { a = n; b = m; }
    int get_a() { return a; }
    int get_b() { return b; }
};
int main() {
  samp sampArray[4][2] = {
    samp(1, 2), samp(3, 4), samp(5, 6), samp(7, 8),
    samp(9, 10), samp(11, 12), samp(13, 14), samp(15, 16)
  };
  for(int i=0; i<4; i++) {
    cout << sampArray[i][0].get_a() << ' ';
    cout << sampArray[i][0].get_b() << "\n";
    cout << sampArray[i][1].get_a() << ' ';
    cout << sampArray[i][1].get_b() << "\n";
  }
  cout << "\n“;  return 0;
}
```

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
```

# *Покажчики на об'єкти*

Щоб оголосити покажчик на об'єкт, використовується той же синтаксис, як і у випадку оголошення покажчиків на значення інших типів. Аналогічно структурам, щоб одержати доступ до окремого члена об'єкта через покажчик на цей об'єкт, необхідно використовувати оператор "стрілка".

```
#include <iostream>
using namespace std;
class P_example {
    int num;
  public:
    void set_num(int val) {num = val;}
    void show_num();
};
void P_example::show_num() {
  cout << num << "\n";
}
int main() {
  P_example ob, *p; // Оголошуємо об'єкт і покажчик на такі об’єкти.
  ob.set_num(1);
  ob.show_num();
  p = &ob; // Присвоюємо покажчику p адресу об'єкта ob.
  p->show_num();
  return 0;
}
```

Покажчик інкрементується або декрементується так, щоб завжди вказувати на наступний або попередній елемент базового типу відповідно (на наступний або попередній об'єкт).

```
#include <iostream>
using namespace std;
class P_example {
    int num;
  public:
    void set_num(int val) {num = val;}
    void show_num();
};
void P_example::show_num() {
  cout << num << "\n";
}
int main() {
  P_example ob[2], *p;
  ob[0].set_num(10); ob[1].set_num(20); // прямий доступ до об'єктів
  p = &ob[0];  // (або p = ob;) Одержуємо покажчик на перший елемент.
 p->show_num();   // Відображаємо значення ob[0] через покажчик.
  p++;         // Переходимо до наступного об'єкта.
 p->show_num();   // Відображаємо значення ob[1] через покажчик.
  p--;         // Повертаємося до попереднього об'єкта.
  p->show_num();  // Снову відображаємо значення елемента ob[0].
  return 0;
}
```

```
10
20
10
```

# *Посилання на об'єкти*

На об'єкти можна посилатися в такий же спосіб, як і на значення будь-якого іншого типу. Для цього не існує ніяких спеціальних інструкцій або обмежень.

Використання посилань на об'єкти дозволяє справлятися з деякими специфічними проблемами, які зустрічаються при використанні класів.